5

CONCLUSIONS AND FUTURE WORK

You're bound to be unhappy if you optimize everything.

— Donald Knuth

The growing adoption of WebAssembly requires hardening techniques. This thesis contributes to this effort with a comprehensive set of methods and tools for Software Diversification in WebAssembly. We introduce three technical contributions in this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs. In this chapter, we summarize the technical contributions of this dissertation, including an overview of the empirical findings of our research. Finally, we discuss future research directions in WebAssembly Software Diversification.

5.1 Summary of technical contributions

This thesis expands the field of Software Diversification within WebAssembly by implementing two distinct methods: compiler-based and binary-based approaches. Taking source code and LLVM bitcode as input, the compiler-based method generates WebAssembly variants. It uses enumerative synthesis and SMT solvers to produce numerous functionally equivalent variants. Importantly, these generated variants can be converted into multivariant binaries, thus enabling execution path randomization. Our compiler-based approach specializes in producing high-preservation variants. However, the use of SMT solvers for functional verification lowers the diversification speed when compared with the binary-based method. Furthermore, this method can only modify the code and function sections of WebAssembly binaries.

Moreover, our binary-based strategy uses random e-graph traversals to create variants. This approach eliminates the need for modifications to existing compilers, ensuring compatibility with all existing WebAssembly binaries. Additionally, it offers a swift, efficient and novel method for generating variants through inexpensive random e-graph traversals. Consequently, our binary-based approach can produce variants at a scale of at least one order of magnitude larger than our compiler-based approach. The binary-based method can generate

variants by transforming any segment of the Wasm binary. However, the preservation of the generated variants is lower than the compiler-based approach.

We have developed three open-source tools that are publicly accessible: CROW, MEWE, and WASM-MUTATE. CROW and MEWE explore a compiler-based approach, whereas WASM-MUTATE employs a method based on binary. These tools automate the process of diversification, thereby increasing their practicality for deployment. At present, WASM-MUTATE is integrated into the wasmtime project[1] to improve testing. Our tools are complementary, providing combined utility. For instance, when the source code for a WebAssembly binary is unavailable, WASM-MUTATE offers an efficient solution for the generation of code variants. On the other hand, CROW and MEWE are particularly suited for scenarios that require a high level of variant preservation. Finally, one can use CROW and MEWE to generate a set of variants, which can then serve as rewriting rules for WASM-MUTATE. Moreover, when practitioners need to quickly generate variants, they could employ WASM-MUTATE, despite a potential decrease in the preservation of variants.

5.2 Key results of the thesis

We demonstrate the practical application of Offensive Software Diversification in WebAssembly. In particular, we diversify 33 WebAssembly cryptomalware automatically, generating numerous variants. These variants successfully evade detection by state-of-the-art malware detection systems. Our research confirms the existence of opportunities for the malware detection community to strengthen the automatic detection of cryptojacking WebAssembly malware. Specifically, developers can improve the detection of WebAssembly malware by using multiple malware oracles. Additionally, these practitioners could employ feedback-guided diversification to identify specific transformations their implementation is susceptible to. For instance, our study found that the addition of arbitrary custom sections to WebAssembly binaries is a highly effective transformation for evading detection. This logic also applies to other transformations, such as adding unreachable code, another effective method for evading detection.

Moreover, our techniques enhance overall security from a Defensive Software Diversification perspective. We facilitate the deployment of unique, diversified and hardened WebAssembly binaries. As previously demonstrated, WebAssembly variants produced by our tools exhibit improved resistance to side-channel attacks. Our tools generate variants by modifying malicious code patterns such as embedded timers used to conduct timing side-channel attacks. Simultaneously, they can produce variants that introduce noise into the execution side-channels of the original program, while altering the memory layout of the JITed code generated by the host engine.

[1] `https://github.com/bytecodealliance/wasm-tools`

Remarkably, we ensure the rapid transformation of WebAssembly binaries, creating thousands of variants in a matter of minutes. This swift generation of variants is particularly advantageous in highly dynamic scenarios such as FaaS and CDN platforms. In this work, we do not discuss this case in depth. Yet, we have empirically tested the effectiveness of moving target defense techniques [37]. These tests were conducted on the Fastly edge computing platform. In this scenario, we incorporate multivariant executions[37]. Fastly can redeploy a WebAssembly binary across its 73 data centers worldwide in 13 seconds on average. This enables the practical deployment of a unique variant per node using our tools. However, a 13-second window may still pose a risk despite each node potentially hosting a distinct WebAssembly variant. To mitigate this, one could use multivariant binaries, invoking a unique variant every time the node is invoked. Our tools can generate dozens of unique variants every few seconds, each serving as a multivariant binary packaging thousands of other variants. This illustrates the real-world application of Defensive Software Diversification to a WebAssembly standalone scenario.

5.3 Future Work

Along with this dissertation, we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several avenues for future research. In the following, we outline three concrete directions.

5.3.1 Data augmentation for Machine Learning on WebAssembly programs

Compared to established environments, the WebAssembly ecosystem is relatively new. This makes the collection of WebAssembly program samples notably expensive [45, 154]. According to a recent study by Hilbig et al., the global count of WebAssembly binaries is approximately 20,000 [45]. This number is small when contrasted with the massive repositories of 1.5 million and 1.7 million packages in npm and PyPI, respectively. Intriguingly, this study also discloses that half of these WebAssembly binaries originated from our tools and public repositories, suggesting that the actual count of unique, real-world WebAssembly programs is just over 10000. This scarcity of WebAssembly datasets presents considerable obstacles for machine learning analysis tools, which typically need substantial data volumes for effective training and calibration [155]. Software diversification serves as a pivotal strategy to address this limitation by simulating a broad range of potential software behaviors and scenarios. Specifically, the augmentation of the WebAssembly dataset can be achieved through it.

Augmentation of program datasets has proven effective in enhancing the accuracy of classification models [156, 157, 158]. In general, data augmentation can improve model performance by increasing the volume of training examples via data synthesis. Moreover, it might expose edge cases and rare conditions,

thus enabling the development of better defenses against unforeseen cases. In the case of malware evasion, this approach might harden the robustness of detection systems. Furthermore, this strategy might facilitate the identification and reduction of biases within WebAssembly program datasets. Finally, our tools provide comprehensive details on the variant generation process. This allows us to define and use more precise metrics between programs and variants to train a model for variant detection [159]. For instance, with the e-graphs in WASM-MUTATE, we can easily establish a metric to measure the distance between two programs. To sum up, harnessing our tools to enhance the training of models within the WebAssembly ecosystem might be a promising research area.

5.3.2 Improving WebAssembly malware detection via canonicalization

Malware detection is a well-known problem in the field of computer security, as outlined in works like Cohen's 1987 study on computer viruses [150]. This issue is exacerbated in environments where predictability is high and malware is expected to be replicated identically across multiple victims. In such scenarios, attackers can exploit this predictability to their advantage. For example, malicious actors could craft functionally equivalent malware to evade detection by malware detection systems. Indeed, our research has shown that employing Software Diversification can be an effective method for evading malware detection systems. This technique involves creating varied versions of a program, thereby reducing its predictability and making detection more challenging.

A future research area could be to tackle the challenge of refining the precision of malware detection systems. This can be achieved by evaluating the effectiveness of program normalization [160]. This strategy might involve transforming a program into a standardized or "canonical" form [161]. A malware detection system in a pre-existing database is more likely to detect the canonical form. Just as a program can be transformed into multiple functionally equivalent variants, the inverse process is also possible. In other words, two functionally equivalent programs can be transformed into the same original program.

Two concrete strategies could be explored. First, employing WASM-MUTATE for program normalization. By reusing the e-graph data structures to use the shortest possible replacements, one can secure the canonical representation of the input program. Although normalization methods are not new, previous studies have grounded malware detection on the normalization of lifted code [162, 163]. WebAssembly does not need to be lifted, given that its binary format is innately platform-agnostic. Thus, one can directly normalize the WebAssembly binary. Secondly, one can pre-compile WebAssembly binaries at a minimal cost. Specifically, a WebAssembly binary could initially undergo JIT compilation into machine code. Overall, either of these two strategies aims to substantially reduce the number of malware variants that need consideration, thereby easing the task of classifiers in detecting harmful software.

5.3.3 Oneshot Diversification

Oneshot diversification denotes the automatic creation of a unique software program variant with each installation or distribution. This procedure entails systematic alterations to the original program. The objective is to ensure that each software copy is distinctive from all others. Contrary to randomization, oneshot diversification usually happens during the software's distribution or installation phase. The term "oneshot" describes the diversification's static nature following its one-time implementation per installation. Once used, the diversified program is discarded. In summary, oneshot diversification bolsters security and heightens reliability by diminishing the predictability of software. We therefore plan to investigate the feasibility of one-shot diversification in WebAssembly. However, we foresee several challenges, particularly the optimization of our previously presented tools.

In the context of WebAssembly, this process presents particularly challenging aspects since it is highly dynamic [17]. For instance, within a browser context, we need to ensure the WebAssembly binary varies not only per browser instance but also per tab process (webpage tab). In the backend, we must guarantee that the WebAssembly binary is unique per cold spawn [50, 63]. Hence, it becomes necessary to ensure that the diversification process is both rapid and efficient, capable of generating a vast number of variants within a brief timespan. Specifically, this entails producing millions of unique and diverse variants every second.

In addition to swift variant generation, a targeted diversification process is also necessary. For example, as shown in Chapter 4, feedback-guided diversification can quickly produce variants with specific objectives, such as evading malware. Therefore, while we diversify, we need to be able to discard those variants that offer fewer benefits based on custom feedback functions. For example, this process might require the inclusion of concepts like performance impact, variant's distribution, and variant's management. Performance impact, in particular, needs careful evaluation, given that WebAssembly is often used for applications sensitive to performance. Furthermore, distributing and managing diversified WebAssembly modules could become complex, especially due to the lack of a standard for WebAssembly module management or registry [12]. Besides, this complexity includes managing updates and ensuring compatibility across all diversified instances.